Assembling a High-Productivity DSL for Computational Fluid Dynamics

Sandra Macià
Barcelona Supercomputing Center
sandra.macia@bsc.es

Pedro J. Martínez-Ferrer
Barcelona Supercomputing Center
pedro.martinez-ferrer@bsc.es

Sergi Mateo
Barcelona Supercomputing Center
sergi.mateo@bsc.es

Vicenç Beltran
Barcelona Supercomputing Center
vicenc.beltran@bsc.es

Eduard Ayguadé
Barcelona Supercomputing Center
eduard.ayguade@bsc.es

ABSTRACT

As we move towards exascale computing, an abstraction for effective parallel computation is increasingly needed to overcome the maintainability and portability of scientific applications while ensuring the efficient and full exploitation of high-performance systems. These circumstances require computer and domain scientists to work jointly toward a productive working environment. Domain specific languages address this challenge by abstracting the high-level application layer from the final, complex parallel low-level code. Saiph is an innovative domain specific language designed to reduce the work of computational fluid dynamics domain experts to an unambiguous and straightforward transcription of their problem equations. The high-level language, domain-specific compiler and underlying library are enhanced to make applications developed by scientists intuitive. Additions and improvements are presented, designed for the significant advantage of running computational fluid dynamics applications on different machines with no porting or maintenance issues. Numerical methods and parallel strategies are independently added at the library level covering the explicit finite differences resolution of a vast range of problems. Depending on the application, a specific parallel resolution is automatically derived and applied within Saiph, freeing the user from decisions related to numerical methods or parallel executions while ensuring suitable computations. Through a list of benchmarks, we demonstrate the utility and productivity of the Saiph high-level language together with the correctness and performance of the underlying parallel numerical algorithms.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; • **Computing methodologies** → *Parallel computing methodologies*; • **Applied computing** → *Physical sciences and engineering*.

KEYWORDS

DSL, CFD, HPC, FDM

1 INTRODUCTION

High-performance computing (HPC) has a central role in scientific research. An essential part of this research is based on simulation of domain-specific applications. Traditionally, domain scientists combine their knowledge with numerical methods and any general purpose language (GPL) to write their scientific applications and solve their problems numerically. However, HPC is evolving rapidly: distributed machines, many-cores processors, accelerators and emerging architectures constitute today's complex and heterogeneous supercomputers. As we move towards exascale computing, HPC users require deeper expertise in computer science and parallel programming models to fully utilise and exploit heterogeneous machines efficiently. Under this scenario, scientists find themselves out of their domain of expertise and productivity decreases dramatically. At the same time, the application's maintainability and portability are becoming a problem. Parallel numerical methods and algorithms are traditionally embedded in the application. In consequence, software or hardware modifications usually lead to a tedious work of rewriting, annotating and tuning. Therefore, a disruptive research strategy becomes necessary to develop maintainable and reusable software that exploits current and future HPC resources effectively.

Domain-specific languages (DSLs) constitute a good alternative to GPLs to tackle the challenges derived from HPC and exascale computing [20]. This separation of concerns through high-level abstraction layers involves a synergy of experts from the different parts of the HPC scene. On the one hand, domain experts describe their applications unambiguously using a high-level, near mathematical specification with all the details required to formalise the whole simulation description. On the other hand, computer scientists are in charge of the numerical methods and the complexities related to the parallel execution. This approach breaks the programming complexity into loosely-coupled specific development layers binding each researcher to his domain of expertise.

Applied to the computational fluid dynamics (CFD) domain, Saiph[19] is a DSL intended to solve the physical equations typically found in CFD engineering problems on HPC environments. Through its modular design, Saiph dissociates the high-level application component where partial differential equations (PDEs) describing physical phenomena are defined, from the low-level execution component where appropriate numerical methods and execution details are determined using domain knowledge. This way, algorithms and parallel strategies are not embedded at the application level anymore. Abstracting over numerical methods, algorithms and hardware provide the application with a stable abstraction layer. CFD applications written with Saiph are independent. Thus they can be easily ported between architectures. Numerical and parallel implementations are application-independent, prone to be reused and tunable for the efficient exploitation of different computer architectures. Saiph is currently based on the explicit Finite Difference Method (FDM) supporting both inter and intra-node

parallelism using MPI[12] and OpenMP[5] programming models. However, its modular design can be enhanced with other numerical methods and parallel strategies targetting different hardware while covering a broader range of CFD problems, giving the potential for a long-term contribution. The DSL development platform also has the potential to be extended to target new domains in other areas of scientific relevance (electromagnetism, relativity, etc.).

Taking advantage of the malleability of Saiph's framework, the contribution of this work is the development of Saiph's internals to provide a productive, explicit FDM, CFD working tool. From the high-level language to the final code the DSL internals are developed ensuring productivity, correctness and performance through (i) automatic validation of equations, (ii) non-uniform mesh support, (iii) operators specialisation, (iv) *Gmsh* input mesh support and (v) parametric binaries. Generic, high-order and stable (vi) spatial schemes, (vii) TVD time schemes and (viii) specific convection schemes, are developed to cover the explicit FDM resolution of a vast selection of CFD applications. Finally, (ix) the automatic application of the suitable numerical parallel method depending on the CFD scenario, enhance the DSL usability.

This paper is structured as follows. Section 2 presents the related work, a literature review of the state of the art and the primary motivation of Saiph. Section 3 revises the DSL design. Section 4 details the developed Saiph's internals making the DSL a productive working tool. The numerical methods and parallel strategies for large applicability of the DSL are presented in Section 5 and 6, along with the corresponding domain-specific optimisations. Finally, Section 7 evaluates the resulting tool and conclusions are given in Section 8.

2 RELATED WORK

The vast majority of state-of-the-art CFD software used today in scientific and industrial applications cannot fully take advantage of current and future HPC hardware. CFD software developers usually fail to keep up the pace with trends in HPC hardware as it is becoming harder to write and maintain performing CFD software with GPLs. An example of this can be found in the widely used open-source CFD library OpenFOAM [15], which proposes its own language syntax to represent physical equations. This library suffers from performance bottlenecks [7] so does not appear to be an ideal candidate for exascale computing. Regarding DSLs, popular projects are proposing different combinations of features such as high-level syntax, specific code generation, numerical methods, portability and parallelism strategies. FiPy [13], OPS/OP2/PyOP2 [22, 24, 25], Liszt [8] and FEniCS [18] are good examples of DSLs employed for the specific resolution of CFD problems. Both FiPy and PyOP2 are based on the widely used Python language; the former relies on the finite volume method (FVM) and can be regarded as a smaller and simpler alternative to OpenFOAM, whilst the later is based on the finite element method (FEM) and focuses more on the performance that can be provided by the OPS/OP2 framework. However, OPS/OP2 parallelism of user code requires the scientist to modify their application and write additional serial code with specific calls to the OPS/OP2 API library. Although this can be seen as a feature, it forces users to modify its application making it difficult to be maintained or ported.

On the other hand, Liszt is a DSL that allows specific domain optimisations but lacks abstraction and high-level syntax. Users have the responsibility to manually discretise their physical equations forcing them to work at the numerical level, besides the domain level. The language is designed for code portability across heterogeneous platforms and provides features for distributed and shared-memory parallelism that should be applied by users to ensure that the Liszt compiler can infer data dependencies automatically.

The approach provided by FEniCS consists of a complete simulation infrastructure for many real-world problems. It relies upon expressing the PDEs at the mathematical level using high-level Python and C++ interfaces. This tool uses a top-level abstraction layer which defines a high-level language for the specification of FEM algorithms allowing users to express the problem in terms of PDEs and leaving the parallel MPI implementation details to a lower-level library. FEniCS can run on multiple platforms from laptops to HPC clusters. Although FEniCS is a compelling solution for a large number of complex problems, it is not at all meant to be used by scientists without deep expertise in numerical methods.

Saiph promises to solve the significant drawbacks found in other DSLs by the innovative combination of the above list of features. It offers a CFD high-level syntax hiding numerical and parallel complexities. It is based on the FDM and automatically combines suitable numerical methods and parallel strategies for the resolution of CFD problems. Its key innovative features are: (i) unprecedented, beyond-state-of-the-art high-level syntax not requiring any knowledge on numerical methods or programming, (ii) automatic numerical resolution and correctness, (iii) automatic parallel resolution, (iv) domain-specific optimisations, (v) design focused on maximum performance for HPC and emergent exascale applications and (vi) reusable and easily enhanced modular design.

The resulting DSL can have a significant impact on the CFD scientific community with the real potential to attract the interest of a large number of users.

3 UNDERLYING DESIGN

Saiph is designed to be simple, efficient, safe and largely applicable. Saiph reduces the work of the domain expert by carrying out the numerical resolution and automatically mapping the resulting application onto different HPC hardware as schematically illustrated in Figure 1 with a CFD equation example.

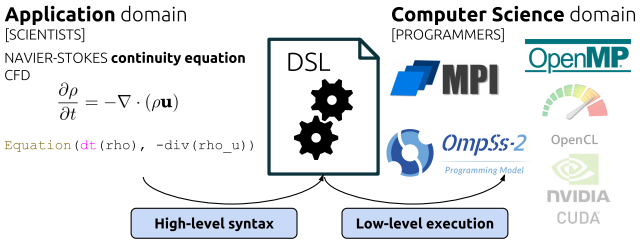

Figure 1: Saiph outer design.

Internally, the DSL is designed to have two main layers, the compiler and the library as illustrated in Figure 2. Productivity and performance are respectively faced in each of them. This separation

eases tool development and allows each layer to be enhanced and reusable. Saiph's modular design guarantees that any new component or optimisation implemented in the DSL platform can be used separately thus maximising its return on investment.

The high-level domain-oriented syntax and the domain-specific optimisations are defined and implemented at the compiler layer. At this layer, the Saiph's source-to-source compiler translates, specialises and optimises the input code. This compiler is embedded in the Scala language [23] using the Lightweight Modular Staging (LMS) [26] as a DSL development platform and the Scala Virtualized Compiler [21]. At the second layer, the Saiph C++ library takes care of the numerical and auxiliary methods and the parallel strategies. The use of different parallel programming models permits the final code to run on a multitude of platforms, from laptops to heterogeneous HPC clusters.

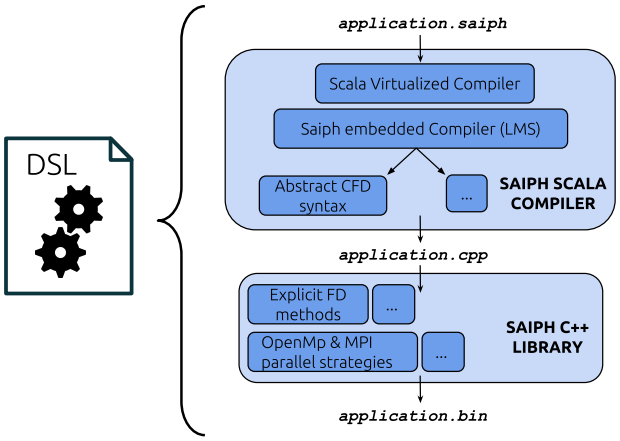

Figure 2: Underlying design and technologies used during the compilation process of a Saiph application.

Currently, the DSL offers a complete high-level syntax for an unambiguous and straightforward transcription of CFD problems into Saiph code as well as specific optimisations, at the first compiler layer. Regarding the C++ library, a set of high-order explicit FDM algorithms has been implemented along with MPI, OmpSs [9] and OpenMP programming models and parallelisation strategies. However, due to its modular nature, Saiph has the potential to be a versatile production tool; back-ends with other programming models annotations and kernels can be easily added to map computations into different architectures such as GPUs or FPGAs. Implicit methods, finite elements or volumes, etc. can enhance the library layer. What is more, an entirely new syntax can be defined and added at the compiler layer enabling work on other domains or even coupled physics simulations.

4 BOOSTING PRODUCTIVITY

We aim to boost the productivity of Saiph domain expert users. The generic high-level syntax, usability and interaction provided by the tool, fit out a fast an intuitive computational modelling process and a productive working environment. Users can give and retrieve information to and from Saiph in an efficient and comfortable natural way, suitable for CFD requirements and tools. The DSL compiler, offering an internal code specialisation and early detection of errors from the problem definition, releases the users from decisions related to numerical methods and eases an error-free application encoding.

4.1 Language

The language is the ultimate utensil for the user's development and represents the origin of the domain-specific information used by the DSL internals. Hence, it is a source of productivity enhancement at different levels.

4.1.1 High-level syntax. Saiph has a high-level syntax to unambiguously define a complete system of PDEs characterising a CFD physical problem [19]. This scientific syntax allows specifying the problem by directly translating the on-paper physical problem definition without requiring expertise on programming for supercomputers or numerical methods. From the spatial domain definition to the system of equations, physical magnitudes and fields are defined, initialised and related through collective and continuous spatiotemporal functions and operations. Units and dimensions are specified for each of the components. Saiph supports several boundary conditions, source terms, vector equations and the most required components encountered on typical CFD problems. Code 1 presents Saiph's application code for the simulation of a 2D convection-diffusion problem with source term.

Use-case 1. *Advection-diffusion problem*

The spatial domain considered is a square of dimensions $0 \leq x \leq 1$, $0 \leq y \leq 1$ *meters, periodic in the x-direction, in SI units*

Governing equation

$$\frac{\partial T}{\partial t} = -\boldsymbol{v} \cdot \nabla T + d \cdot \nabla^2 T + S(t) \tag{1}$$

Constants

$$\boldsymbol{v} = (0m/s, 1m/s)$$
$$d = 0.001m^2/s$$

Initial condition

$$T(x, y) = e^{-\left(\frac{x^2}{2} + \frac{y^2}{2}\right)} K$$

Boundary conditions

$$\begin{cases} T(0, y) = T(1, y) \\ T(x, 0) = 0K \\ \frac{\partial T(x,1)}{\partial y} = 0K/m \end{cases}$$

Heat source at $(x, y) = (0.1m, 0.2m)$

$$S(t) = \begin{cases} 300K/s & at \mid t < 0.2s \\ 0K/s & at \mid t \geq 0.2s \end{cases}$$

```
// 2D Spatial domain definition
val mesh = CartesianMesh(1 * Meters, 1 * Meters)
// Space discretisation factors
mesh.discretize(1 * mm, 1 * mm)
// Mesh boundary condition
mesh.setPeriodic(DirX)
// Constants
val v = Constant(Speed)("Velocity", Vector(0 * m/s, 1 * m/s),
     "Vx, Vy")
```

```
val d = Constant(Meters2/Seconds)("Diff coeff", 0.001 * m2/s)
// f(x, y) defining a Gaussian pulse, ICs
val T = Variable(Temperature)("T", mesh, (x, y, z) => {
  exp(-(pow(x, 2)/2.0 + pow(y, 2)/2.0)) * K })
// T Boundary conditions
T.setDirichlet(CFaceYMIN, 0 * Kelvins)
T.setNeumann(CfaceYMAX, 0 * Kelvins/Meters)
// Heat source
// At (x=0.1m, y= 0.2m), a f(t) defines an inlet heat flux.
val S = Source(Temperature/Seconds)(mesh, (0.1*m, 0.2*m),
   (t) => { if(t < 0.2*s) 300.0*K/s
            else 0.0*K/s })
// Governing equation
val AdvDiffEq = Equation(dt(T), -v*grad(T) + d*lapla(T) + S)
// Problem Definition
val prob = Problem(mesh)(AdvDiffEq)
// Solver
Stepper(prob, DT, NSTEPS, IntegrationMethod.RK3)("ADVDIFF2D",
     OutputFormat.VTI, SamplingMethod.Flush)
```

Code 1: 2D Advection-diffusion Saiph application code.

The whole modelling process is achieved with a few lines of code transcribing the original problem, the number of time-steps to simulate and the spatiotemporal discretisation factors.

4.1.2 Internal validations. The user provides units and dimensions of the variables of its system. Using this information, Saiph internally checks that definitions, initialisation of variables, boundary conditions, operations and equations are valid. Descriptive compilation errors prevent the user from performing illegal problem specifications allowing him to identify and correct mistakes.

Different units can be utilised to represent the same magnitude: miles and kilometres to express a length or Kelvins and Celsius degrees for temperatures. Internally, units are stored in the International System of Units (SI), and numerical values are accordingly transformed to ensure correct computations while applying unity-type-check. Equations and some operators have left, and right-hand side expressions with units that must match, otherwise Saiph would emit an error. For this test, the DSL creates an internal tree representation of each equation, whose leaves are the variables and constants of the problem. Units are checked and computed for each node of the equation tree, bottom up, according to the internally defined operator rules on units.

Regarding spatial dimensions, some operators are restricted to be applied to scalar or vector fields. In contrast, the right-hand side of the equation is only allowed to be a scalar careless of the nature of the left-hand side, which can be either a scalar or vector. This syntax restriction allows defining vector equations efficiently and unambiguously through the use of the component or subscript operators. Similarly to units, node's dimensions, characterised from the nature of the tree leaves and the operator rules, are checked and computed for each equation tree node, in a bottom-up way.

Figure 3 illustrates the way Saiph tests the validity of equation (2) by checking dimensions and units of their child and characterising nodes. For instance, the *addition node* emits and error whether the nature of its operands differs on dimensions or units. If the match occurs, the *addition node* takes the characteristics of its children. The *gradient node* on the other hand, is defined as a vector node and only accepts scalar child nodes. Since it performs a spatial derivative, its units are computed by dividing the units of its only operand, by meters.

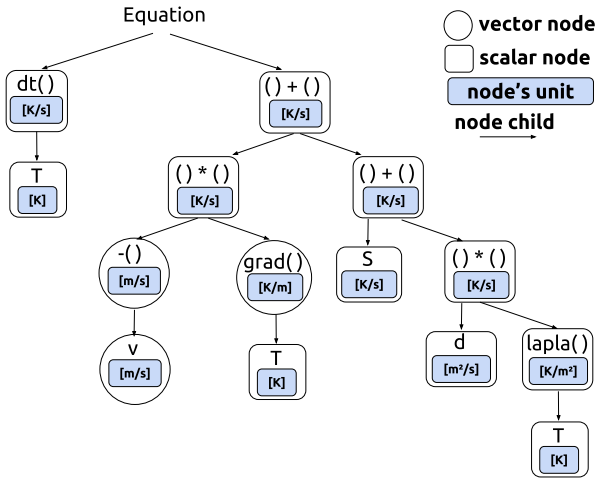

Figure 3: Units and dimensions validation on operators and equation.

4.1.3 External meshes. Spatial domains are not always easily described through continuous functions. However, more generic geometries with finner discretised regions (using non-uniform spatial discretisation factors) or non-regular spatial regions can be set through the use of external meshes as input for Saiph. The already discretised input meshes are defined giving the path to the mesh file and the base unit of its length dimension (by default, the length unit is *Meters*).

```
def Cartesian(path: string, baseUnit: Length) : Cartesian
```

Code 2: External mesh constructor.

The use of external meshes eases the spatial domain definition. It allows for the discretisation refinement of zones of interest and defines spatial regions with label identifiers that can be later on used within the application, for the initialisation or boundary conditions of constants and variables, as illustrated in Code 3.

```
val T = Variable(Temperature)("Temp", mesh, 300* K)
T.setIC("airfoil", 0*K)
T.setNeumann("outlet", 0*K/m)
```

Code 3: Use of mesh labels as spatial regions.

4.1.4 Operators specialisation. Herein we present Saiph optimisations which are specific of the CFD domain. They occur at the first stage of the compilation process and aim to ensure the correct use of the underlying numerical methods. From the analysis of the problem and its equations, the Saiph compiler generates specific IR operators nodes.

Saiph's syntax has a complete set of mathematical operators available to combine variables and constants, either scalars or vectors, to build the system of equations. Each of them has been internally overloaded to perform a suitable operation depending on the nature of the problem and the operands involved.

Spatial differentiation operators are specifically generated depending on the nature of the mesh. The spatial operator emitted for non-uniform meshes involves the coefficient derived from the chain rule $\partial\epsilon/\partial x$ needed to take into account the variations of the spatial discretisation factor Δx.

Other operators such as the *product operator* $()*()$, are translated depending on the dimensions of their operands. Any combination is possible, so the operations are unambiguously defined. Code 4 presents the operation specialisation: a *dot product* is generated when both operands are vectors, a *scalar product* for scalar operands and a *scaling factor* otherwise.

```
def infix_*(x: vector, y: vector) : scalar // Dot product
def infix_*(x: scalar, y: scalar) : scalar // Scalar product
def infix_*(x: vector, y: scalar) : vector // Scaling factor
```

Code 4: *Product* operator overload.

The DSL is also capable of applying numerical domain-specific optimisations. Some of those optimisations aim to ensure numerical stability through the analysis of the problem equations. For example, the convective term of equation (2) is identified and a specific *convective gradient* operator emitted. The specific spatial differentiation operator involves skew, upwind differentiation schemes to avoid numerical instabilities instead of default central schemes. Operator overload permits the users to code a CFD application unambiguously. As in a hand-written manner, users would not need to take care of using the precise, particular operators for each case. Instead, specification and stability optimisations are internally performed ensuring a productive, correct and stable utilisation of the underlying numerical methods.

4.2 *Gmsh* support

The DSL enhancement has also been devoted to easing the task of either getting data from the users and giving information to them.

Externally generated meshes can be defined and discretised using the popular mesh generator, Gmsh [10], to, later on, be used as input within the Saiph application. This significant contribution permits simulating both uniform and non-uniform meshes. First of all, the Gmsh output mesh is converted to a *.saiph.mesh* file by a provided Perl script. From the *.saiph.msh* input file the mesh information can be read in parallel by Saiph: number of dimensions 1D, 2D or 3D, axes coordinate arrays X, Y, Z and spatial regions predefined and labelled in Gmsh. Figure 4 illustrates this input mesh process.

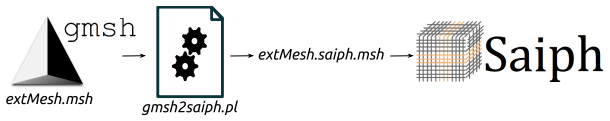

Figure 4: External mesh input process.

4.3 Working methodology

After the compilation process, Saiph generates a binary specific to the application and ready to be executed in parallel. Usually, CFD domain experts work across a range of numerical and execution parameters of the CFD application under study. To provide a productive research methodology, the generated binary accepts a bunch of numerical and output parameters as run-time arguments and is currently ready to run sequentially or with a valid parallel configuration making use of inter and intra-node processes.

The run-time numerical and output arguments are the input mesh file, its length and time discretisation factors, the number of time-steps, the order of the time-integration method, the spatial accuracy, output simulation name, format and sampling method.

Regarding the parallel execution, the configuration is also set at run-time through the environment variables and job scripts of the machine under use. The underlying resources involved are specified by the user through the desired number of inter and intra-processes.

Once the modelling and compilation process is accomplished, an exhaustive parametric study and performance analysis can be done over the same application with no need to rewrite or recompile it.

5 NUMERICAL RESOLUTION

In general, PDE systems describing fluid dynamics cannot be solved analytically. By using numerical methods, continuous functions and equations are spatiotemporally discretised so approximate solutions can be obtained. The internal Saiph CFD library's goal is to give high-order accuracy approximation to flows described by initial and boundary conditions using high-order generic FDM schemes. Special attention on consistency, stability and convergence, covering a large Courant-Friedrichs-Lewy (CFL) range has been taken. At the application level, users select the order of accuracy of spatial and temporal schemes for their applications while numerical resolution and assembly details are internally and automatically handled.

5.1 Spatial schemes

The continuous spatial dimensions are discretised through the use of Cartesian meshes. Spatial differentiation operators are derived from Taylor series expansions and polynomial fitting up to order eight for the approximation of the first and second spatial derivative functions. By default, central schemes with a truncation error of $O(\Delta x^4)$ are used. When using high-order schemes, the approximations for near-boundary nodes require special treatment. Saiph uses adaptive stencils based on forward and backward schemes, biased towards the interior of the solution domain, to maintain the global accuracy order. The coefficients for those approximations have been derived through the method of undetermined coefficients [17] for approximations up to order eight. Similarly, Neumann boundary conditions, derived from spatial differentiation formulas, use the corresponding skewed schemes to avoid the generation and propagation of errors from the boundaries to the rest of the mesh. For non-uniform grids, a set of coefficients $\partial\epsilon/\partial x_i$ derived from the chain rule $\frac{1}{\partial x_i} = \partial\epsilon^{-1}(\frac{\partial\epsilon}{\partial x_i})$ with $\partial\epsilon = cst$ are added to the coefficients of the schemes for the non-uniform specific operators.

Saiph aims to offer a controlled spatial accuracy. Users can preselect the order of accuracy. Differentiation operators, adaptive

stencils and boundary conditions are automatically defined to be consistent with each other and with the given order of accuracy.

5.2 Temporal schemes

Saiph uses explicit time integration methods. Available schemes are the first-order forward Euler method and the Runge-Kutta family of methods, up to order fourth. Explicit methods suffer from instability if the time step is larger than a certain CFL condition number. Runge-Kutta methods are suggested as they effectively use a smaller step size. Moreover, the implemented schemes correspond to the high-order Total Variation Diminishing (TVD) Runge-Kutta methods [11] with better stability when solving hyperbolic conservation equations. The system of algebraic equations defining CFD problems is commonly composed of equations of different nature, time-derivative (as conservation laws) and non-time derivative (as the equation of state) internally stored using tree structures as illustrated in Figure 3. To maintain the order of accuracy, the available time integration methods for first and second PDEs are automatically combined with the resolution of non-time derivative equations. Figure 5 illustrates Saiph work-flow in which the most consuming part corresponds to the *update unknowns* function, going across equation trees at each spatial point.

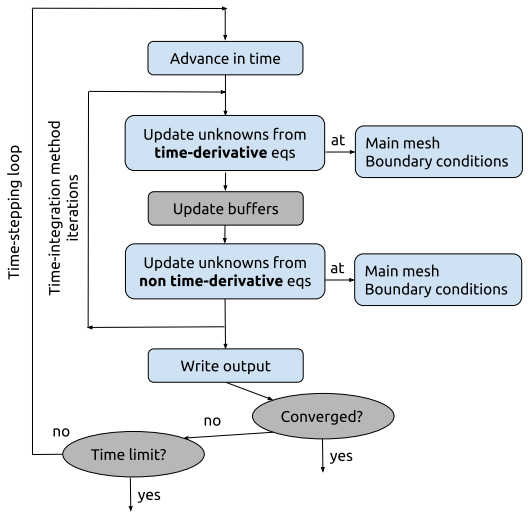

Figure 5: Saiph time-advancing work-flow.

Similarly to the spatial accuracy, the temporal accuracy is set by the user and internally ensured through the coordinated time advancing work-flow.

5.3 Convection schemes

The convective term represents the propagation of a certain quantity in a flow field in the direction of the flow velocity. It is mathematically translated as the dot product of the fluid velocity u and the gradient of the scalar property ϕ being transported $\mathbf{u} \cdot \nabla\phi$.

Saiph takes special care of this term, as it commonly induces numerical instabilities or excessive numerical diffusion incurring in non-physical results or a drop of the global order of accuracy. The DSL can identify this term and compute it according to its requirements. Different schemes are internally applied depending on the uniformity of the mesh, the dimensionality of the problem, the sign and space variability of the velocity vector and the smoothness of the profile of the magnitude being transported. The goal is to output stable and physical results of this term covering a vast range of CFL condition values under those scenarios.

5.3.1 Upwind schemes. To simulate the propagation of information in the direction determined by the velocity vector, adaptive finite difference skewed schemes are used. They compute the gradient of the property being transported and are based on a differentiation biased in the direction determined by the sign of the flow velocity. They are called *upwind* schemes because they skew the derivative computation in the direction the data is "coming from", therefore using points in the direction of convection determined by the sign of the vector velocity components. The available *upwind* schemes in Saiph are the first-order one-point upstream scheme and the third-order Leonard scheme [16], both capable of dealing with non-uniform meshes while maintaining the spatial order of accuracy.

5.3.2 Corner transport upstream. For multi-dimensional convection, the Corner Transport Upstream (CTU) method [6] is applied to correctly combine the convection for each of the propagation directions successively. Saiph implements this strategy whenever the velocity vector has more than one non-zero component, as a local dimension splitting. Using this splitting methodology, the stability range is enlarged since the CFL condition must be ensured in each one of the spatial directions, separately. The CTU method is combined in Saiph with any *upwind* scheme.

5.3.3 Total Value Diminishing constraint. Solutions of the convective term have a non-increasing variation. Numerical methods used to solve this kind of hyperbolic terms must prevent numerical oscillations around flow discontinuities, regardless of the smoothness of the profile being convected. This requirement is translated as the variation of the solution being diminishing. To satisfy the TVD criteria [14], a limiting function is introduced to the convection schemes [2]. The magnitude of the limiting function depends on the smoothness of the profile. It is computed using the ratio of the consecutive gradients so, the TVD constraint has no impact for continuous smooth profiles. Otherwise, it prevents from non-physical instabilities at the expense of a decrease in spatial accuracy.

5.3.4 Method of characteristic lines. The method of characteristic lines [1] is a semi-Lagrangian scheme [3] constituting a good alternative to the previously described methods when dealing with constant propagation velocities. In such scenarios, a one-to-one relationship can be established for each spatial coordinate. The quantities for a certain coordinate x_i are transported to another one x_j at each time-step. With a suitable time-space discretisation where the "simulation velocity" is proportional to the fluid velocity, $\Delta x/\Delta t = k \cdot v$ with $k \in \mathbb{Z}$, the transport can be done by directly copying the values of the transported quantities. For these suitable meshes the method is, therefore, free of truncation errors and non-restrictive with the time-discretisation factor: the new constraint associated with the convection term is given by $CFL = k$ with $k \in \mathbb{Z}$. When the discretisation factor ratio is not proportional to

the constant velocity of the fluid, the method introduces interpolation functions carrying spatial truncation errors. Those functions use values from the neighbour of the spatial coordinate being transported, which does not correspond to a real grid point. Since no explicit integration is used when applying this scheme, the CFL constraint does not apply. Moreover, this scheme does not generate numerical instabilities, even for discontinuous profiles.

5.3.5 Combining schemes. As well as for space and time schemes, Saiph deals with the combination of convection methods. From the term detection, the most suitable scheme is applied. Constant velocities are early detected, and the method of characteristic lines used. For that, the convection term is removed from its original equation. At each time step, the rest of the equation is regularly computed, and the transported quantities added to the result. In such scenarios, the convection term does not restrict the CFL anymore. Thus, larger CFL ranges are supported. Regarding non-constant velocities scenarios, the one-point upstream or the Leonard schemes are selected depending on the spatial accuracy set by the user. The velocity characteristics and the profiles being transported determine the application of CTU and TVD methods. Local dimension splitting and limiter functions ensure a stable computation of the convection term. Convection of continuous and non-continuous profiles with multi-dimensional velocities covering a large CFL range is automatically applied and combined in Saiph internals.

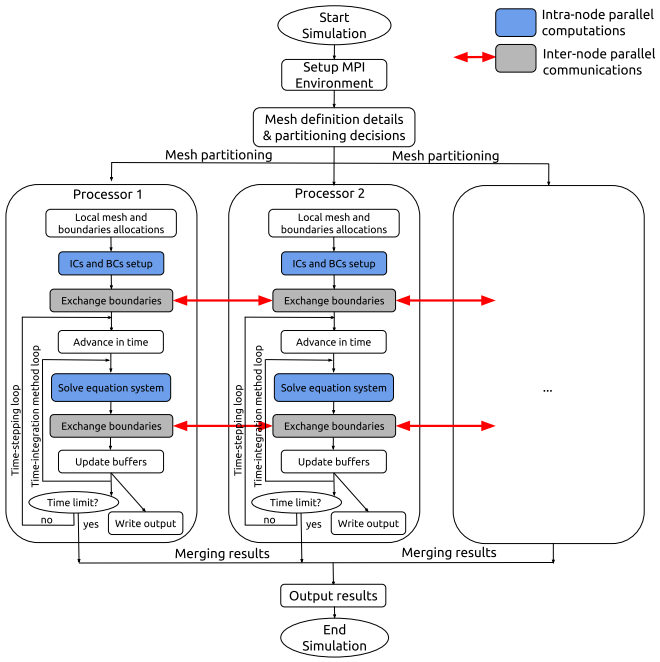

Figure 6: Saiph parallel work-flow.

6 PERFORMANCE FEATURES

Saiph has been designed to take advantage of HPC by applying knowledge of the specific domain. Hence, all the DSL features have underlying parallel support allowing Saiph applications to run on HPC environments. Performance optimisations are also transparently applied to produce a specific efficient parallel execution. The modular tool permits to enhance performance by easily adding optimisations maintaining the high-level application unchanged.

6.1 Parallel approach

Inter and intra-node parallelism are harmoniously combined in Saiph internals. For the inter-node parallelism, the mesh is partitioned by the last dimension. Regarding intra-node parallelism, partial differential equations are integrated in parallel at every time-step. Differential equations are solved in parallel for all the points of the local mesh. Figure 6 illustrates Saiph's parallel work-flow running on an HPC cluster.

6.2 Perfomance optimisations

The DSL compiler also applies performance optimisations such as computations and storage reduction and data locality improvements to enhance the efficiency of the final generated code.

Nested operations can be simplified by identifying the operators and the nature of the operands involved. Constants and variables can be redefined depending on their initialisation and use [19].

Another example of performance optimisation is the distribution of the mesh across the available MPI processes. The mesh partition is done through a single dimension (the "last" one, e.g. z-dimension in a 3D app) so that boundaries are contiguous in memory and thus efficiently communicated. To ensure maximising the number of partitions and minimising the communication cost, Saiph can internally permute spatial dimensions. The distribution occurs along the mesh direction with the highest workload.

7 EVALUATION

We evaluate Saiph's productivity, output results, numerical methods and scalability for various CFD applications. The chosen benchmarks combine different phenomena and scenarios: 1D, 2D and 3D uniform and non-uniform meshes are tested for continuous and discontinuous fluid profiles. External sources, different boundary conditions and flow discontinuities have also been combined through different systems of equations, coupling first, second and non-derivative equations. Mathematical specifications, Saiph codes, evaluations and output results can be looked up at https://github.com/EulerStokes/CFD-benchmarkApp. Here we present two applications illustrating Saiph utilisation and performance.

7.1 Gaussian pulse convection

This application illustrates the use of external non-uniform meshes and the error-free simulation of flow transport.

7.1.1 Problem specification. Code 5 presents the Saiph implementation of the 2D convection of a Gaussian pulse.

Use-case 2. *2D Gaussian Pulse Convection*

The spatial domain considered is a square of dimensions $-6 \leq x \leq 6$, $-6 \leq y \leq 6$ *meters, periodic in the x and y-directions, in SI units*

Governing equation

$$\frac{\partial T}{\partial t} = -\boldsymbol{v} \cdot \nabla T \tag{2}$$

Constants

$$\boldsymbol{v} = (1, 0)$$

Initial condition

$$T(x, y) = e^{-\left(\frac{x^2}{2}+\frac{y^2}{2}\right)}K$$

The application is coded in Saiph as follows.

```
// Cartesian mesh input
val mesh = CartesianMesh("path/mesh.saiph.msh", Meters)
mesh.setPeriodic(DirX)
mesh.setPeriodic(DirY)
// ICs
val T = Variable(Temperature)("T", mesh, (x, y, z) => {
  exp(-(pow(x, 2)/2.0 + pow(y, 2)/2.0)) * K })
val u = Constant(m/s)("u", Vector(1*m/s, 0*m/s), ("X", "Y"))
// Equation definition
val convEq = Equation(dt(T), -u * grad(T))
// Problem definition
val probConv = Problem(mesh)(convEq)
// Time parameters
val Dt = 0.04*s
val nsteps = 300
// Solver
Stepper(probConv, 0.04*s, 900, IntegrationMethod.Euler)("2
     DGaussianPulse", OutputFormat.Binary, SamplingMethod.
     Periodic, 1)
```

Code 5: 2D Gaussian Pulse application code.

7.1.2 Simulation results. The problem has been set with a 2D external non-uniform Cartesian mesh using $249 * 27$ grid points. The Gaussian pulse suffers a constant net transport along the x-direction, so the initial profiles are conserved during the simulation. Several convection cycles have been simulated using Euler's time-integration method. By analysing the convective velocity and the smoothness of the profile, Saiph has applied the method of characteristics lines for the computation of the convection term. Figure 7 shows the final results which have been validated against the initial flow conditions. This simulation has no truncation error $\boldsymbol{L_2 = 0K}$ according to the fact that the DSL identifies the problem as linear and applies the analytic-solution method of characteristic lines.

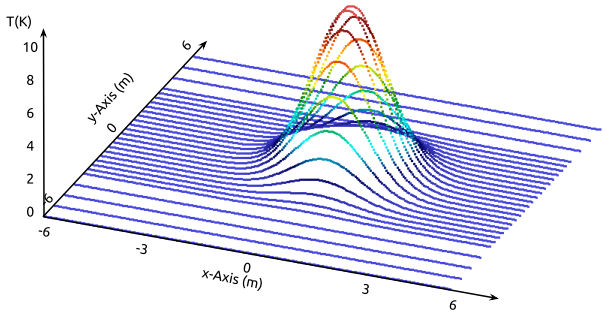

Figure 7: Output grid points for the 2D Gaussian Pulse convection on a non-uniform mesh, after n-convection cycles.

7.2 Inviscid vortex convection problem

For this second test case, we present the specification and resolution of a system of coupled equations. A numerical and scalability evaluation is performed over this application example.

7.2.1 Problem specification. At the user level, Code 6 illustrates the direct map from the high-level syntax to the domain-specific constructs. Around 30 lines of code are enough to encode the following mathematical problem specification.

Use-case 3. *2D Euler Equations - Inviscid Vortex Convection*

The spatial domain considered is a square of dimensions $0 \leq x \leq 10$, $0 \leq y \leq 10$ *meters, periodic in the x and y-directions (SI units).*

Governing equation: Euler equations

$$\frac{\partial \rho}{\partial t} = -\boldsymbol{v} \cdot \nabla \rho - \rho \boldsymbol{\nabla} \cdot \boldsymbol{v} \tag{3}$$

$$\frac{\partial (\rho v_i)}{\partial t} = -\boldsymbol{v} \cdot \nabla (\rho v_i) - (\rho v_i) \boldsymbol{\nabla} \cdot \boldsymbol{v} - (\nabla p)_i \tag{4}$$

$$\frac{\partial (\rho E)}{\partial t} = -\boldsymbol{v} \cdot \nabla (\rho E) - (\rho E) \boldsymbol{\nabla} \cdot \boldsymbol{v} - \boldsymbol{v} \cdot \nabla p - p \boldsymbol{\nabla} \cdot \boldsymbol{v} \tag{5}$$

$$p = (\gamma - 1)\left(\rho E - \frac{1}{2}\rho v^2\right) \tag{6}$$

Adiabatic index $\gamma = 1.4$
Vortex strength $b = 0.5$
Vortex initial center $(x_c, y_c) = (5, 5)$
Distance from the vortex center $r = \left[\left((x - x_c)^2 + (y - y_c)^2\right)\right]^{1/2}$
Problem unknown and initial conditions

$$\rho = \left[1 - \frac{(\gamma - 1)b^2}{8\gamma\pi^2} e^{1-r^2}\right]^{\frac{1}{\gamma - 1}}$$

$$v_x = \frac{b}{2\pi} e^{\frac{1}{2}(1-r^2)}(y - y_c)$$

$$v_y = 0.1 - \frac{b}{2\pi} e^{\frac{1}{2}(1-r^2)}(x - x_c)$$

$$p = 1$$

The application is coded in Saiph as follows.

```
// 2D uniform regular Cartesian mesh and BCs
val mesh = CartesianMesh(10*m, 10*m)
mesh.discretize(0.01953125*m, 0.01953125*m)
mesh.setPeriodic(DirX)
mesh.setPeriodic(DirY)
// Parameters
val adiabaticIdx = 1.4
val vStrenght = 0.5
val v_x = Vector(1.0 * m/s, 0.0 * m/s)
val v_y = Vector(0.0 * m/s, 1.0 * m/s)
// Variables and constants
val gamma_minus1 = Constant(Unitless)("AdiabaticIndex -1",
   (adiabaticIdx -1)*Unitless)
val v = Variable(Speed)("Velocity", mesh, (x, y, z) => {
  ((vStrenght / (2.0 * PI)) * exp(0.5 * (1.0-(pow((x.value -
       5.0), 2) + pow((y.value - 5.0), 2)))) * (y.value - 5.0))
        * v_x + (1.0 - ((vStrenght / (2.0 * PI)) * exp(0.5 *
       (1.0 - (pow((x.value - 5.0), 2) + pow((y.value - 5.0),
       2)))) * (x.value - 5.0))) * v_y }, ("X", "Y"))
val p = Variable(Pressure)("Pressure", mesh, (x, y, z) => {
```

```
17    pow(pow(1 - (((adiabaticIdx -1.0) * vStrenght * vStrenght) /
          (8.0 * adiabaticIdx * PI * PI)) * exp(1.0 - (pow((x.
          value - 5.0), 2) + pow((y.value - 5.0), 2))), (1.0 / (
          adiabaticIdx - 1.0)))*Unitless, adiabaticIdx) * Pascals
           })
18  val rho = Variable(kg/m3)("Density", mesh, 1.0*kg/m3)
19  val rhoU = Variable(kg/m2/s)("Rho*U", mesh, Vector(0 * kg/m2/s,
          0 * kg/m2/s), ("X", "Y"))
20  val rhoE = Variable(kg/m/s2)("Rho*E", mesh, 0.0 * kg/m/s2)
21  // Initialization equations
22  val init_rhoe = Equation(rhoE, (p/(adiabaticIdx -1*Unitless))
         + (0.5*Unitless) * rho * v * v)
23  val init_rhou = Equation(rhoU, rho * v.i)
24  // Euler Equations
25  val density = Equation(dt(rho), -v * grad(rho) - rho * div(v))
26  val momentum = Equation(dt(rhoU), -v * grad(rhoU.i) - rhoU.i *
          div(v) - grad(p).i)
27  val energy = Equation(dt(rhoE), -v * grad(rhoE) - rhoE * div(v
         ) - v * grad(p) - p * div(v))
28  val velocity = Equation(v, rhoU.i/rho)
29  val state = Equation(p, (adiabaticIdx -1*Unitless)*(rhoE -
         (0.5*Unitless)*rhoU*v))
30  // Problem definition. (Init equations)(Governing equations)
31  val prob = Problem(mesh)(init_rhoe, init_rhou)(density,
         momentum, energy, velocity, state)
32  // Solver
33  Stepper(prob, 3.125*ms, 3200, IntegrationMethod.RK3)("
         InviscidcVortex", OutputFormat.VTK, SamplingMethod.
         FinalState)
```

Code 6: 2D Inviscid Vortex Saiph application code.

7.2.2 Simulation results. The application has been set with a 2D uniform Cartesian mesh using $512 * 512$ grid points. The vortex suffers a net transport on the y-direction, so the initial profiles are conserved along the simulation. Initial conditions can thus be seen as analytic solutions of the problem. 3200 time-steps have been computed to complete a full convection cycle. A default spatial accuracy of $O(\Delta x^4)$ is set, while the TVD-RK3 time-integration method ($O(\Delta t^3)$) is selected at the user level. By analysing the convection velocity and the smoothness of the profiles the Leonard scheme ($O(\Delta x^3)$) combined with CTU and TVD methods is applied for the computation of the convection terms. Figure 8 shows the final results validated against the initial flow conditions.

After one convection cycle, the L_2 norm of the pressure quantity, taking the initial conditions as reference results is $\boldsymbol{L_2 = 4 \cdot 10^{-6}}$Pa.

7.2.3 Evaluation of numerical methods. The aim of this analysis is to calculate the order of accuracy of the numerical methods used. We run the application several times, changing discretisation factors and the time-step parameter. At each run, the values of Δx and Δt are divided by two. The L_2 norm is computed taking initial and final results, always reporting the error at one convection cycle. Figure 9 illustrates the tendency of the error depending on the spatial discretisation factor. The logarithmic fitting $log(y(x)) = a + 2.3 * log(x)$ shows a slope of 2.3. This number corresponds to the real order of the global truncation error which differs from the theoretical one ($O(\Delta x^3)$). The difference is explained by the use of the TVD method when dealing with convection phenomena.

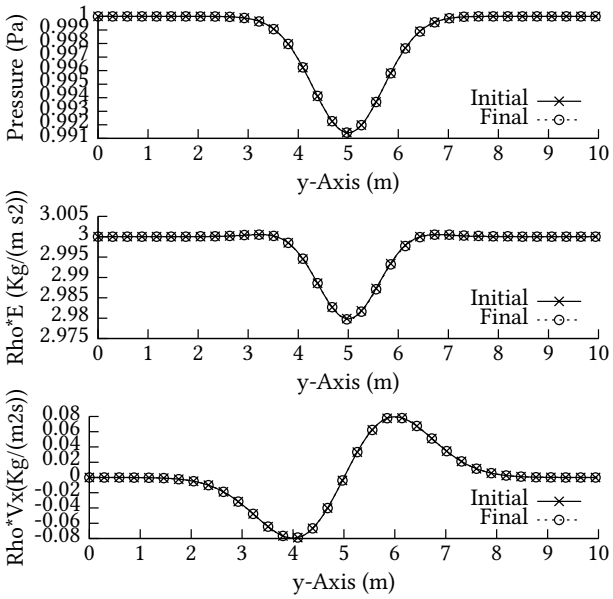

Figure 8: Variable profiles after one convection cycle for the Inviscid Vortex Convection application.

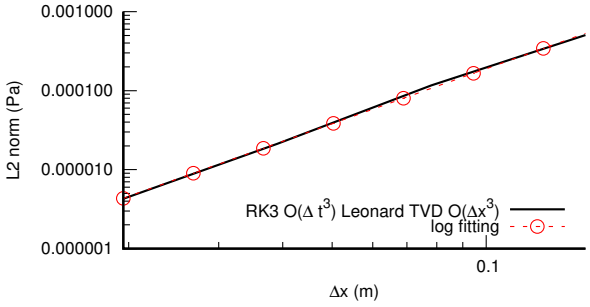

Figure 9: Order of accuracy analysis.

This mechanism induces the slight drop of the order of accuracy favouring the stability of the result.

7.2.4 Scalability evaluation. To evaluate the parallel scalability of Saiph, we run the application on BSC's MareNostrum 4 supercomputer [4]. This supercomputer integrates 3,456 general purpose nodes with a total of 165, 888 processor cores and 390 terabytes of main memory. Each compute node is equipped with two Intel Xeon Platinum 8160 CPU sockets with 24 cores each. A high-speed Omnipath network connects all the components. We analyse the strong scalability of the inviscid vortex convection problem involving the computation of 3.9 million grid points - 2D mesh of 512 points per direction, 5 unknowns and 3 RK3-intermediate-steps: $(512 \cdot 512) \cdot (5 \cdot 3) \approx 3.9M$ - and 3200 time-steps to perform a full convection cycle of 10s. To leverage the hybrid parallelisation strategy, each simulation has been performed using OpenMPI, binding each MPI process to a node making use of its 48 cores through OpenMP threads. A linear speedup has been obtained up to 192

cores. Scalability results are shown in Table 1, taking as base-line the execution time of the application when running on a single node with 48 OpenMP threads. The quasi-linear speedup degrades as more processors involved due to the increase of the replicated data (boundaries) and their communications.

#nodes	#cores	Speedup
1	48	1
2	96	2.01
4	192	4.0
8	384	7.8
16	768	15.5
32	1536	29.8
48	2304	45.2

Table 1: Saiph scalability results

The aim of this evaluation is to demonstrate the parallel nature of Saiph's generated code. Although the design and parallel capability to automatically run on HPC environments are fully functional, efforts have not yet been devoted to boost the efficiency of the code. To this end, approaches are planned to be developed and added to the library layer such as generation of kernels to run on GPUs, SIMD exploitation, locality enhancement, computation and communication overlap, etc. A deep parallel and efficiency study is contemplated as future work.

8 CONCLUSIONS

The work presented has been explicitly developed for boosting the productivity of CFD scientists on HPC environments by improving the degree of abstraction and usability of Saiph's high-level syntax to suit the productivity demands of the scientific community. Saiph internally applies and combines suitable parallel numerical schemes for the stable computation of a large set of CFD problems that can be solved through explicit FDMs. We have shown that Saiph outputs accurate results of CFD applications while having the ability to exploit modern supercomputers. We believe that Saiph contributes toward an efficient expression of CFD applications that dramatically reduces the demand of knowledge on numerical methods and parallel programming and, at the same time, eases portability and maintainability tasks. Likewise, Saiph is presented as a supporting infrastructure in which improvements and enhancements at different levels can be easily added to enlarge the applicability range. The DSLs philosophy arises as a promising research methodology for facing the exascale era.

ACKNOWLEDGMENTS

This work is supported by the Ministry of Economy of Spain through Severo Ochoa Center of Excellence Program (SEV-2015-0493), by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P) and by the Generalitat de Catalunya (2017-SGR-1481).

REFERENCES

[1] Muhammad Adeel Ajaib. 2013. Numerical Methods and Causality in Physics. *arXiv preprint arXiv:1302.5601* (2013).

[2] Roumen Anguelov, Jean M-S Lubuma, and Froduald Minani. 2010. Total variation diminishing nonstandard finite difference schemes for conservation laws. *Mathematical and Computer Modelling* 51, 3-4 (2010), 160–166.

[3] JR Bates and A McDonald. 1982. Multiply-upstream, semi-Lagrangian advective schemes: Analysis and application to a multi-level primitive equation model. *Monthly Weather Review* 110, 12 (1982), 1831–1842.

[4] BSC-CNS. [n. d.]. *MareNostrum*. Available at https://www.bsc.es/marenostrum.

[5] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. 2008. *Using OpenMP: portable shared memory parallel programming*. Vol. 10. MIT press.

[6] Phillip Colella. 1990. Multidimensional upwind methods for hyperbolic conservation laws. *J. Comput. Phys.* 87, 1 (1990), 171–200.

[7] Massimiliano Culpo. 2011. Current bottlenecks in the scalability of OpenFOAM on massively parallel clusters. *PRACE* (2011).

[8] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 9.

[9] Alejandro Duran, Eduard Ayguade, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel processing letters* 21, 2 (2011).

[10] Christophe Geuzaine and Jean-François Remacle. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre-and post-processing facilities. *International journal for numerical methods in engineering* 79, 11 (2009), 1309–1331.

[11] Sigal Gottlieb and Chi-Wang Shu. 1998. Total variation diminishing Runge-Kutta schemes. *Mathematics of computation of the American Mathematical Society* 67, 221 (1998), 73–85.

[12] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface*. Vol. 1. MIT press.

[13] Jonathan E Guyer, Daniel Wheeler, and James A Warren. 2009. FiPy: Partial differential equations with Python. *Computing in Science & Engineering* 11, 3 (2009).

[14] Ami Harten. 1983. High resolution schemes for hyperbolic conservation laws. *Journal of computational physics* 49, 3 (1983), 357–393.

[15] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. 2007. OpenFOAM: A C++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, Vol. 1000. IUC Dubrovnik, Croatia, 1–20.

[16] Liu Jianchun, Gary A Pope, and Kamy Sepehrnoori. 1995. A high-resolution finite-difference scheme for nonuniform grids. *Applied mathematical modelling* 19, 3 (1995), 162–172.

[17] Randall J LeVeque. 2004. Finite volume methods for hyperbolic problems. *Cambridge Texts in Applied Mathematics* 39, 1 (2004), 88–89.

[18] Anders Logg, Kent-Andre Mardal, and Garth Wells. 2012. *Automated solution of differential equations by the finite element method: The FEniCS book*. Vol. 84. Springer Science & Business Media.

[19] Sandra Macià, Sergi Mateo, Pedro J Martínez-Ferrer, Vicenç Beltran, Daniel Mira, and Eduard Ayguadé. 2018. Saiph: Towards a DSL for High-Performance Computational Fluid Dynamics. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*. ACM, 6.

[20] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005).

[21] Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. 2012. Scala-virtualized *(PEPM '12)*. ACM, New York, NY, USA, 117–120. https://doi.org/10.1145/2103746.2103769

[22] Gihan R Mudalige, IZ Reguly, and Michael B Giles. 2016. Auto-vectorizing a large-scale production unstructured-mesh CFD application. In *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*. ACM, 5.

[23] Martin Odersky, Lex Spoon, and Bill Venners. 2010. *Programming in Scala, Second Edition*. Artima.

[24] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. 2012. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In *SCC, 2012 SC Companion:*. IEEE, 1116–1123.

[25] István Z Reguly, Gihan R Mudalige, Michael B Giles, Dan Curran, and Simon McIntosh-Smith. 2014. The OPS domain specific abstraction for multi-block structured grid computations. In *WOLFHPC, 2014 Fourth International Workshop on*. IEEE, 58–67.

[26] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs *(GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314